To appear in: Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html

# Scoped Dynamic Rewrite Rules

Eelco Visser

*Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. http://www.cs.uu.nl/people/visser, visser@acm.org*

**Abstract**

The applicability of term rewriting to program transformation is limited by the lack of control over rule application and by the context-free nature of rewrite rules. The first problem is addressed by languages supporting user-definable rewriting strategies. This paper addresses the second problem by extending rewriting strategies with scoped dynamic rewrite rules. Dynamic rules are generated at run-time and can access variables available from their definition context. Rules generated within a rule scope are automatically retracted at the end of that scope. The technique is illustrated by means of several program tranformations: bound variable renaming, function inlining, and dead function elimination.

## 1 Introduction

Rewrite rules provide a good formalism for expressing program transformations. A rewrite rule defines a local transformation derived from an algebraic equality of programs. There are two problems associated with the application of standard term rewriting techniques to program transformation: the need to intertwine rules and strategies in order to control the application of rewrite rules and the context-free nature of rewrite rules.

### *1.1 Exhaustive Application of Rules*

Exhaustive application of all rules to the entire abstract syntax tree of a program is not adequate for most transformation problems. The system of rewrite rules expressing basic transformations is often non-confluent and/or non-terminating.

An ad hoc solution that is often used is to encode control over the application of rules into the rules themselves by introducing additional function symbols. This intertwining of rules and strategy obscures the underlying program equalities, incurs a programming penalty in the form of rules that define a traversal through the abstract syntax tree, and disables the reuse of rules in different transformations.

The paradigm of programmable rewriting strategies solves the problem of control over the application of rules while maintaining the separation of rules and strategies. A strategy is a little program that makes a selection from the available rules and defines the order and position in the tree for applying the rules. Thus rules remain pure, are not intertwined with the strategy, and can be reused in multiple transformations.

Support for strategies is provided by a number of transformation systems in various forms. In TAMPR [4] a transformation is organized as a sequence of canonical forms. For each canonical form a tree is normalized with respect to a subset of the rules in the specification. ELAN [3] provides non-deterministic sequential strategies. Stratego [17,15] provides generic primitive traversal operators that can be used to compose generic tree traversal schemas. See [16] for a survey of strategies in program transformation.

## *1.2 Context-free Nature of Rewrite Rules*

The second problem of rewriting is the context-free nature of rewrite rules. A rule has only knowledge of the construct it is transforming. However, transformation problems are often context-sensitive. For example, when inlining a function at a call site, the call is replaced by the body of the function in which the actual parameters have been substituted for the formal parameters. This requires that the formal parameters and the body of the function are known at the call site, but these are only available higher-up in the syntax tree.

There are many similar problems in program transformation, including bound variable renaming, typechecking, constant and copy propagation, and dead code elimination. Although the basic transformations in all these applications can be expressed by means of rewrite rules, they need contextual information.

The usual solution to this problem is to extend the traversal over the tree (be it hand-written or generic) such that it distributes the data needed by transformation rules. For example, traversal functions in ASF+SDF [5] can be declared to have an accumulation parameter in which data can be collected. Language independent definitions of operations such as bound variable renaming in Stratego [15] capture a generic tree traversal schema that takes care of distributing an environment through a tree. The disadvantage of these solutions is that the traversal strategy becomes data heavy instead of just handling control flow. That is, all traversal functions become infected with additional parameters carrying context information. Generic solutions break down when multiple environments are needed, to handle multiple name spaces, for instance.

Another solution is the use of contextual rules [2,17]. Contextual rules combine the context and the local transformation in one rule by using a local traversal that applies a rule that reuses information from the context. The problem with this approach is that it performs an extra traversal over the abstract syntax tree, leading to quadratic complexity in case the contextual rule is applied as part of a traversal over the same tree that the context accesses.

### *1.3 Dynamic Rules*

This paper introduces the extension of rewriting strategies with *scoped dynamic rules*. A dynamic rule is a normal rewrite rule that is generated at run-time and that can access information from its generation context. For example, to define an inliner, a rule that inlines function calls for a specific function can be generated at the point where the function is declared, and used at call sites of the function.

Dynamic rules are first-class. Their application is under control of a normal strategy. Thus dynamic rules can be applied as part of a global tree traversal. Rules can overrule the definition of previously generated rules. To restrict the application of a dynamic rule to a certain part of the tree, the live range of a rule can be determined by rule scopes (Section 3). A rule temporarily overruled in a scope becomes visible again at the end of that scope. To hide rules generated in outer scopes, rules can be undefined (Section 4). Rules from outer scopes can also be permanently overridden (Section 5).

The mechanism of dynamic rules as described in this paper reflects the implementation of dynamic rules in Stratego version 0.6.2 available from `www.stratego-language.org`.

### *1.4 Outline*

Section 2 reviews the basics of rewriting strategies in Stratego. Sections 3 through 5 introduce dynamic rules by means of a number of transformations on Tiger programs: bound variable renaming (Section 3), function inlining (Section 4), and dead function elimination (Section 5). Each of these examples motivates and illustrates an aspect of dynamic rules. Section 6 discusses other applications, related and future work. Section 7 concludes.

## 2 Rewriting Strategies

This section reviews the basics of rewriting strategies in Stratego as far as needed for this paper. See [17] for details of the operational semantics underlying the language. In this paper Tiger, the example language in the compiler construction textbook of Appel [1], is used to illustrate the application of dynamic rules.

### *2.1 Program Representation*

In Stratego, programs to be transformed are expressed as first-order terms. Signatures describe the structure of terms. A term over a signature `S` is either

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in fact(10)
end
```

```
Let([
  FunctionDec([
    FunDec("fact",[FArg("n",Tid("int"))],Some(Tid("int")),
      If(RelOp(LT,Var("n"),Int("1")),
         Int("1"),
         BinOp(MUL,Var("n"),
               Call(Var("fact"),
                    [BinOp(MINUS,Var("n"),Int("1"))]))))])],
  [Call(Var("fact"),[Int("10")])])
```

Fig. 1. Concrete and abstract syntax of a small Tiger program.

a nullary constructor `C` from `S` or the application `C(t1,...,tn)` of an `n`-ary constructor `C` from `S` to terms `ti` over `S`. List notation `[t1,...,tn]` is an abbreviation for terms constructed with the constructors `Cons` and `Nil`.

Figure 2 shows the signature of the abstract syntax of Tiger programs. Tiger is an imperative, first-order language with nested functions. Data are composed using arrays and records from integers and strings. Control flow is determined using if-then-else, while and for. Examples of terms over the Tiger signature are `Var("x")` (the variable `x`), `Call(Var("f"), [Var("x")])` (call of function `f` with argument `x`), and `Let([VarDec("x", None, Int("1"))], [Var("x")])` (the declaration of local variable `x` initialized to the integer constant `1`). Figure 1 gives a small example program and the corresponding abstract syntax representation.

### *2.2 Rewrite Rules*

Rewrite rules express basic transformations on terms. A rewrite rule has the form `L : l -> r`, where `L` is the label of the rule, and the term patterns `l` and `r` are its left-hand side and right-hand side, respectively. A term pattern is either a variable, a nullary constructor `C`, or the application `C(p1,...,pn)` of an `n`-ary constructor `C` to term patterns `pi`. For example,

```
  Fold : BinOp(PLUS, e, Int("0")) -> e
```

is a simple constant folding rule for Tiger expressions.

A rule `L: l -> r` applies to a term `t` when the pattern `l` matches `t`, i.e., when `l` has the same top-level structure as `t`. Applying `L` to `t` has the effect of transforming `t` to the term obtained by replacing the variables in `r` with the subterms of `t` to which they correspond. Actually the basic actions underlying rules are first-class operations in Stratego. The operation `?t` denotes matching against the term pattern `t`, and `!t` denotes building an instantia-

```
module Tiger-Core
signature
  sorts Exp LValue InitField Tdec Dec FunDec FArg Type Field TypeId
  constructors
    Var          : String -> Var
    FieldVar     : LValue * String -> LValue
    Subscript    : LValue * Exp -> LValue

    Int          : String -> Exp
    String       : String -> Exp
    NilExp       : Exp

    Call         : Var * List(Exp) -> Exp
    Record       : TypeId * List(InitField) -> Exp
    InitField    : String * Exp -> InitField
    Array        : TypeId * Exp * Exp -> Exp
    BinOp        : BinOp * Exp * Exp -> Exp
    RelOp        : RelOp * Exp * Exp -> Exp

    Seq          : List(Exp) -> Exp
    Assign       : LValue * Exp -> Exp
    If           : Exp * Exp * Exp -> Exp
    IfThen       : Exp * Exp -> Exp
    While        : Exp * Exp -> Exp
    For          : Var * Exp * Exp * Exp -> Exp
    Break        : Exp
    Let          : List(Dec) * List(Exp) -> Exp

    Tdec         : String * Type -> Tdec
    TypeDec      : List(Tdec) -> Dec
    VarDec       : String * Option(TypeId) * Exp -> Dec
    FunctionDec  : List(FunDec) -> Dec
    FArg         : String * TypeId -> FArg
    FunDec       : String * List(FArg)
                     * Option(TypeId) * Exp -> FunDec

    NameTy       : TypeId -> Type
    RecordTy     : List(Field) -> Type
    Field        : String * TypeId -> Field
    ArrayTy      : TypeId -> Type
    Tid          : String -> TypeId
```

Fig. 2. Signature of the abstract syntax of Tiger

tion of the term pattern `t`. Thus, a rule `L: l -> r` is just syntactic sugar for `L = {x1,...,xn: ?l; !r}`, where `{x1,...,xn: s}` delimits the scope of the pattern variables `x1,...,xn`. The construct `{s}` implicitly makes all free variables in `s` local.

### 2.3 *Rewriting Strategies*

Programmable rewriting strategies provide a mechanism for achieving control over the application of rewrite rules, while avoiding the introduction of new constructors or rules. A rewriting strategy is a program that transforms terms or fails at doing so. In the case of success, the result is a transformed term. In the case of failure, there is no result.

Rewrite rules are just strategies which apply transformations to the roots of terms. Strategies can be combined into more complex strategies by means of Stratego's strategy operators. The *identity* strategy `id` always succeeds and leaves its subject term unchanged. The *failure* strategy `fail` always fails. The *sequential composition* `s1 ; s2` of strategies `s1` and `s2` first attempts to apply `s1` to the subject term. If that succeeds, it applies `s2` to the result; otherwise it fails. The *non-deterministic choice* `s1 + s2` of strategies `s1` and `s2` attempts to apply either `s1` or `s2` to the subject term, but in an unspecified order. It succeeds if either `s1` or `s2` succeeds, and fails otherwise. The *deterministic choice* `s1 <+ s2` of strategies `s1` and `s2` first attempts to apply `s1` to the subject term. Only if `s1` fails, it attempts to apply `s2` to the subject term. If `s1` and `s2` both fail, the choice fails as well. The *recursive closure* `rec x(s)` of a strategy `s` attempts to apply to the subject term the strategy obtained by replacing each occurrence of the variable `x` in `s` by the strategy `rec x(s)`.

A strategy definition `f(x1,...,xn) = s` introduces a new strategy operator `f` parameterized with strategies `x1,...,xn` that applies body `s`.

### 2.4 *Generic Term Traversal*

The strategy combinators just described combine strategies which apply transformation rules to the roots of their subject terms. In order to apply a rule at an internal site of a term (i.e., to a subterm), it is necessary to traverse the term. Stratego defines several primitive operators which expose the direct subterms of a constructor application. These can be combined with the operators described above to define a wide variety of complete term traversals. For the purposes of this paper we restrict the discussion of traversal operators to congruence operators and the `all` operator.

Congruence operators provide one mechanism for term traversal in Stratego. If `C` is an `n`-ary constructor, then the congruence `C(s1,...,sn)` is the strategy that applies only to terms of the form `C(t1,...,tn)`, and works by applying the strategies `si` to the terms `ti`. For example, the congruence `Let(s1,s2)` transforms terms of the form `Let(t1,t2)` into `Let(t1',t2')`, where `t1'` is the result of applying `s1` to `t1`, and similarly for `t2'`. If the application of `si` to `ti` fails for any `i`, then the application of `C(s1,...,sn)` to `C(t1,...,tn)` also fails.

The operator `all(s)` applies `s` to all direct subterms `ti` of a constructor application `C(t1,...,tn)`. It succeeds if and only if all applications to the direct subterms succeed. The resulting term is the constructor application `C(t1',...,tn')` where the `ti'` are the results obtained by applying `s` to the terms `ti`. Note that `all(s)` is the identity on constants, i.e., on constructor applications without children. An example of the use of `all` is the strategy `topdown`, defined as

```
topdown(s) = rec x(s; all(x))
```

The strategy expression `rec x(s; all(x))` specifies that the parameter transformation `s` is first applied to the root of the current subject term. If that succeeds, the strategy is applied recursively to all direct subterms of the term, and, thereby, to all of its subterms. This definition of `topdown` captures the generic notion of a pre-order traversal over a term.

## 3 Bound Variable Renaming

Bound variable renaming is a transformation that replaces bound variables and their corresponding occurrences by new unique names. As a result of the transformation a name is used by at most one binding. This transformation is necessary to prevent free variable capture when substituting expressions under bindings, for example when performing function inlining.

The following transformation illustrates renaming of variable declarations in Tiger programs.

```
let var i := 1
 in (let var i := (i + 2)
      in i
     end + i)
end
```

⇒

```
let var a_0 := 1
 in (let var b_0 := (a_0 + 2)
      in b_0
     end + a_0)
end
```

Note that the `i` used in the initialization of the second declaration is bound by the outer declaration, and that the `i` used after the inner let also refers to the outer declaration. However, the `i` inside the inner let refers to the inner declaration. These issues are clarified by the renamed version on the right.

The abstract syntax representation for the expression on the left is the term:

```
Let([VarDec("i",None,Int("1"))],
    [BinOp(PLUS,
           Let([VarDec("i",None,BinOp(PLUS,Var("i"),Int("2")))],
               [Var("i")]),
           Var("i"))])
```

```
exprename(Let([VarDec(x, t, e1)], e2), rn) =
  Let([VarDec(y, t, exprename(e1, rn))], exprename(e2, (x,y) : rn))
  where new => y
exprename(Var(x), rn) =
  Var(lookup(x, rn))
```

```
exprename(BinOp(op, e1, e2), rn) =
  BinOp(op, exprename(e1, rn), exprename(e2, rn))
```

Fig. 3. Definition of renaming function. The first box contains the essential rules. The second box shows an example of the other rules the definition consists of.

### *3.1 Functional Definition of Renaming*

A conventional implemention of bound variable renaming defines a function `exprename` that recursively visits all nodes of an abstract syntax tree, carrying a renaming table `rn`, which is extended at binding sites and consulted at variable occurrences (Figure 3). Note that for renaming the initializer of the declaration the unextended renaming environment is used. The `new` function generates a new string that is unique in the sense that it does not occur anywhere in the current syntax tree.

The disadvantage of this implementation is that the function has to explicitly visit all tree nodes, even those (such as `BinOp`) that are not variables or do not bind variables. Thus, the definition of a renaming function has a rule for each constructor following the schema

```
exprename(C(e1, ..., en), rn) =
  C(exprename(e1, rn), ..., exprename(en, rn))
```

For a full definition of renaming of Tiger programs, 6 essential renaming rules and 17 additional rules are required. For real languages the ratio of essential rules over additional rules is likely to decrease.

### *3.2 Renaming using Rewrite Rules*

Generic traversals make it possible to avoid the overhead of defining traversals of constructs not involved in the transformation at hand. The recursive rename function above can be expressed essentially by a topdown traversal

```
exprename = topdown(try(RenameVarDec + RenameVar))
```

This strategy traverses an abstract syntax tree, and at each subtree tries to apply one of the rules `RenameVarDec` or `RenameVar`. The operator `try` is defined as `try(s) = s <+ id`, i.e., it tries to apply a transformation `s`, but if that fails returns the original term. Only rules for constructs that are actually changed need to be provided.

The first rule renames the binding variable in a variable declaration by generating a new name:

```
RenameVarDec :
  Let([VarDec(x, t, e1)], e2) -> Let([VarDec(y, t, e1)], e2)
  where new => y
```

The second rule renames an occurrence of `Var(x)` to `Var(y)`:

```
RenameVar :
  Var(x) -> Var(y)
```

But here is the catch: in rule `RenameVar` the intention is not to rename just any variable to any other variable, but to rename an occurrence of a bound variable to its new name generated at its binding site, i.e., in rule `RenameVarDec`.

## *3.3 Generating Renaming Rules*

The rule for renaming a variable thus depends on the renaming of the corresponding binding construct. This dependency can be expressed using dynamic rules. `RenameVarDec` can be reformulated such that it generates a renaming rule for the variable that is bound by the declaration construct:

```
RenameVarDec :
  Let([VarDec(x, t, e1)], e2) -> Let([VarDec(y, t, e1)], e2)
  where new => y
      ; rules(RenameVar : Var(x) -> Var(y))
```

The dynamic rule declaration `rules(RenameVar : Var(x) -> Var(y))` in the condition of `RenameVarDec` generates an instance of the `RenameVar` rule that inherits the values of the meta-variables `x` and `y` in its context.

As an example of the operation of this dynamic rule generation, consider the application of `RenameVarDec` to the term

```
Let([VarDec("i", None, Int("1"))], ... )
```

When matching the left-hand side of the rule against this term, the variable `x` is bound to the string `"i"`. Subsequently, a new unique string, say `"a_0"`, is generated by `new`, and bound to the variable `y`. In the context of these bindings the dynamic rule `RenameVar` is created, resulting in the generation of the rule:

```
RenameVar : Var("i") -> Var("a_0")
```

Finally, the renamed variable declaration is produced:

```
Let([VarDec("a_0", None, Int("1"))], ... )
```

While further traversing the tree, each occurrence of `Var("i")` to which the dynamically generated rule `RenameVar` is applied is replaced by `Var("a_0")`, while other (free) variables are not affected.

### *3.4 Limiting the Scope of Generated Rules*

The topdown renaming strategy using `RenameVarDec` and `RenameVar` as defined above is not quite right, yet. If we apply it to our example expression, we get:

```
let var i := 1
 in (let var i := (i + 2)
      in i
     end + i)
end
```

⇒

```
let var a_0 := 1
 in (let var b_0 := (b_0 + 2)
      in b_0
     end + b_0)
end
```

The renaming rule generated for the inner declaration of `i` overrides the rule for the outer declaration. That is correct for the occurrences of `i` inside the inner let, but the rule also still applies after the inner let.

This problem suggests that it should be possible to retract generated rules after the scope in which they are valid ends. This is exactly what the *rule scope* `{| lab : s |}` achieves. A rule with label `lab` that is generated while executing `s` is automatically removed at the end of the scope. Thus, any rule that was overridden by rules generated inside the scope becomes visible again after the scope. Recall that `exprename` was defined as

```
exprename = topdown(try(RenameVarDec + RenameVar))
```

where the definition of `topdown` is:

```
topdown(s) = rec x(s; all(x))
```

By redefining the renaming strategy as

```
exprename =
  rec r({| RenameVar :
           try(RenameVarDec + RenameVar); all(r)
        |})
```

a `RenameVar` rule generated by a variable declaration is automatically removed after exiting the scope. It is necessary to inline the definition of `topdown` since the scope of the generated renaming rules should include the traversal of the subterms with `all(r)`.

Consider the effect of the new strategy on the example and note that `i` after the inner let is now correctly renamed:

```
let var i := 1
 in (let var i := (i + 2)
      in i
     end + i)
end
```

⇒

```
let var a_0 := 1
 in (let var b_0 := (b_0 + 2)
      in b_0
     end + a_0)
end
```

However, the renaming is still not correct, since the `i` in the initializer is not renamed correctly.

To also correctly treat initializers of variable declarations the traversal should be adapted such that variables in initializers are renamed before generating a new renaming rule. That is what the following strategy achieves. The congruence `Let([VarDec(id,id,r)],id)` visits the initializer of a variable declaration, while the congruence `Let([VarDec(id,id,id)],r)` only visits the body of the `Let`:

```
exprename =
  rec r(try(Let([VarDec(id,id,r)],id));
        {| RenameVar :
           try(RenameVarDec + RenameVar);
           (Let([VarDec(id,id,id)],r) <+ all(r))
        |})
```

The choice `(Let([VarDec(id,id,id)],r) <+ all(r))` adapts the generic traversal just for the case of a variable declaration.

### *3.5 Abstracting over Rule Generation*

Since there are several constructs that bind variables in Tiger it is unattractive to repeat the code for generating renaming rules for every binding construct. This can be avoided by creating a rule that transforms a name into a fresh name and at the same time generates a renaming rule.

Figure 4 defines a full-fledged renaming strategy for Tiger programs covering all binding constructs, and also renaming type identifiers, thus dealing with two name spaces simultaneously. The renaming rules for the binding constructs call rule `NewVar` to generate a new name and a corresponding variable renaming rule. The rule is defined as follows:

```
NewVar : x -> y
   where new => y; rules(RenameVar : Var(x) -> Var(y))
```

The renamer defined using this rule renames all variables, even if a variable name was already unique. For some applications it is useful to rename as few variables as possible, for instance, when the result should be readable by a programmer. One approach is to rename only those variables that clash with outer bindings. In our running example this approach has the following effect:

```
let var i := 1
 in (let var i := (i + 2)
      in i
     end + i)
end
```

⇒

```
let var i := 1
 in (let var a_0 := (i + 2)
      in a_0
     end + i)
end
```

This can be achieved by only generating a new name for variables that already exist in an outer scope. The following rule tries to apply `RenameVar` to the variable `x`. If that succeeds the variable was already declared in an outer scope. In that case a new variable is generated. Otherwise the original variable name

```
module Tiger-Rename
imports Tiger dynamic-rules lib
strategies
  exprename =
    rec r(try(Let([VarDec(id,id,r)],id))
          ; {| RenameVar, RenameTid :
               try(RenameDeclaration + RenameArgs + RenameFor
                   + RenameVar + RenameTid);
               (Let([VarDec(id,id,id)],r) <+ all(r))
            |})

  RenameDeclaration =
    Let([RenameVarDec + FunctionDec(map(RenameFun))
         + TypeDec(map(RenameTdec))], id)

  RenameVarDec :
    VarDec(x, t, e) -> VarDec(y, t, e)
    where <NewVar> x => y

  RenameFun :
    FunDec(f, xs, t, e) -> FunDec(g, xs, t, e)
    where <NewVar> f => g

  RenameArgs :
    FunDec(f, xs, t, e) -> FunDec(f, ys, t, e)
    where <map(FArg(NewVar,id))> xs => ys

  RenameFor :
    For(Var(x), e1, e2, e3) -> For(Var(y), e1, e2, e3)
    where <NewVar> x => y

  RenameTdec :
    Tdec(x, t) -> Tdec(y, t)
    where <NewTid> x => y
```

Fig. 4. Renaming of bound variables and type names

is used.

```
NewVar : x -> y
   where (<RenameVar> Var(x); new <+ !x) => y
       ; rules(RenameVar : Var(x) -> Var(y))
```

### *3.6 Summary*

In this section we have seen how context-dependent rewrite rules can be generated using the `rules(...)` construct using information from the context in which they are defined. Subsequent generation of rules overrides previously generated rules. A rule scope `{| lab : s |}` limits the live range of a rule generated by `s` to the scope. Dynamic rules for several name spaces (e.g., variables and types), can be generated at the same time. Dynamic rules can be used like static rules in a generic traversal of the tree structure. Thus only relevant tree nodes are visited explicitly, other nodes are traversed implicitly.

## 4 Function Inlining

Function inlining is a transformation that replaces a function call by the body of the function in which the actual parameters have been substituted for the formal parameters. For example, consider the following simple example, in which a call to the `sqr` function is replaced by its body.

```
let function sqr(x : int) =
      (x * x)
 in sqr((3 + y))
end
```

$\Rightarrow$

```
let function sqr(x : int) =
      (x * x)
in let var a_0 : int := (3 + y)
   in (a_0 * a_0)
   end
end
```

Note that the replacement introduces local variables to bind the actual parameters to the (renamed) formal parameters. This is necessary since Tiger is an imperative language. Simply substituting the actual parameters for the formal parameters could lead to duplication of work or even to errors because of intervening assignments. Further optimizations such as constant and copy propagation can get rid of the local declarations if possible.

The function inlining transformation is expressed by the `InlineFun` rule that is defined as follows:

```
InlineFun :
  Call(Var(f),es) -> Let(ds, e)
  where <exprename-all> fdec => FunDec(_, xs, t, e)
      ; <zip(BindVar)> (xs, es) => ds
```

A function call is replaced with a let expression that has the body of the function declaration `e` as its body. Furthermore, the let introduces a list of variable declarations corresponding to the formal parameters `xs` of the function declaration (after renaming the declaration). The local variables `xs` are bound to the actual parameters `es` by zipping together the lists of formals and actuals and building a variable declaration using rule `BindVar`:

```
BindVar :
  (FArg(x,t), e) -> VarDec(x, Some(t), e)
```

The variable `fdec` in the rule should be bound to the original function declaration of `f`. This information is not normally available at the call site of the function. By generating rule `InlineFun` dynamically when encountering a

```
module Tiger-Inline
imports Tiger Tiger-Rename
strategies
  inline(s1,s2) =
    rec x(s1; try(Let([VarDec(id,id,x)],id))
          ; {| InlineFun :
               (Declare; Let([VarDec(id,id,id)] <+ x, id)
                       ; Declare; Let(id, x)
                <+ all(x))
             ; s2
            |})

  Declare =
    Let([FunctionDec(map(DeclareFun <+ UnDeclareFun))
         + UnDeclareVars], id)
    + UnDeclareVars

  DeclareFun =
    ?fdec@FunDec(f, _, _, _);
    inlineable;
    rules(
      InlineFun :
        Call(Var(f),es) -> Let(ds, e)
        where <exprename-all> fdec => FunDec(_, xs, t, e)
            ; <zip(BindVar)> (xs, es) => ds
    )

  BindVar :
    (FArg(x,t), e) -> VarDec(x, Some(t), e)

  UnDeclareFun =
    ?FunDec(x, _, _, _);
    rules( InlineFun : Call(Var(x),_) -> Undefined )

  UnDeclareVars =
    (?VarDec(x,_,_) + ?For(Var(x),_,_,_));
    rules( InlineFun : Call(Var(x),_) -> Undefined )
```

Fig. 5. Simple inlining strategy

function declaration the necessary information can be passed on to `InlineFun`. In Figure 5 strategy `DeclareFun` generates an inlining rule for a function declaration if it is `inlineable`.

Figure 5 defines a simple inlining strategy `inline` that is parameterized with two transformation strategies `s1` and `s2`. These are transformations to apply on the way down the tree (`s1`) and on the way up (`s2`). An example instantiation could be

```
inline(repeat(InlineFun + Simplify), repeat(Simplify))
```

that inlines functions on the way down and simplifies expressions (e.g., constant folding) on the way down and up. The strategy basically comes down to the following:

```
inline(s1,s2) =
  rec x(s1; {| InlineFun : try(Declare); all(x); s2 |})
```

That is, first the `s1` transformation is applied. Then, after entering the scope for the `InlineFun` rule, `Declare` generates inline rules for any local function declarations. Subsequently all subtrees are visited recursively. After that the `s2` transformation is applied. The strategy in Figure 5 deals with the scope rule of variable declarations and regenerates the inline rules after optimizing their bodies such that only already optimized functions are inlined.

### *4.1 Undefining Rules*

Strategy `DeclareFun`, which generates the `InlineFun` rules, only does so when the function declaration is deemed to be `inlineable`. The exact definition of `inlineable` does not matter here; it could be defined using various heuristics based on static or dynamic program analyses. What does matter is the fact that for non-inlineable functions no `InlineFun` rule is generated. If two functions with the same name exist, one shadowing the other, and the outer is inlineable while the inner is not, this could lead to replacing a call with the wrong function body. Thus, it is necessary to prevent inlining rules from outer scopes to creep trough.

Dynamic rules can be declared as *undefined*. The strategy `UnDeclareFun` generates an `InlineFun` rule that is undefined:

```
UnDeclareFun =
  ?Fdec(x, _, _, _);
  rules( InlineFun : Call(Var(x),_) -> Undefined )
```

This rule always fails when called. The effect is to hide any rules from outer scopes for the same function name. The same is done for variable declarations and loop counter variables, since they may shadow function definitions.

An alternative to undefining `InlineFun` for functions that should not be inlined is to compute the `inlineable` condition in the where clause of the generated `InlineFun` rule instead of computing it at generation time. This expensive, however, since it would entail recomputing the condition everytime the rule is called.

## 5 Dead Function Elimination

The purpose of dead code elimination is to remove code fragments from a program that are never used at run-time. Dead function elimination is a special case of dead code elimination in which function declarations are removed if the function being defined is never called. An example of dead function elimination is the following transformation that takes the result of inlining from the previous section and removes the, now unused, `sqr` function:

```
let function sqr(x : int) =
      (x * x)
in let var a_0 : int := (3 + y)
   in (a_0 * a_0)
   end
end
```

⇒

```
let var a_0 : int := (3 + y)
 in (a_0 * a_0)
end
```

Elimination of dead functions requires a traversal over the program to establish whether there are any calls to a function. Figure 6 gives a definition of dead function elimination using dynamic rules. The strategy is to declare each function to be dead by default. The strategy `DeclareDead` defines the rule `IsDead` for a function once its declaration is in scope. When, during the traversal of the syntax tree, a call to a function is encountered, the `IsDead` rule is *undefined* by `NotDead`. On the way out all functions for which rule `IsDead` still succeeds are then eliminated by strategy `Eliminate`, which filters out all functions that should be eliminated. Note that this strategy should be refined in order to eliminate dead (mutually) recursive functions.

### *5.1 Overriding Rules*

This strategy requires a new kind of dynamic rule introduction. Consider a definition of `NotDead` using regular dynamic rules:

```
NotDead =
  ?Call(Var(f),_);
  rules( IsDead : FunDec(f,_,_,_) -> Undefined )
```

This would entail that a new rule `IsDead` would be added for the function called at that position. However, this new rule would be removed as soon as the transformation exits the surrounding scope and the function declaration would still be eliminated.

However, this is not what we want, since the intention of `NotDead` is to change the original rule defined in the scope of the function declaration, rather than to undefine `IsDead` for local purposes. This is achieved by declaring the dynamic rules as `override rules`. The generation of an overriding dynamic rule only succeeds if there was a prior definition of a dynamic rule for the same left-hand side.

## 6 Discussion

### *6.1 Other Applications*

Dynamic rules have been applied succesfully in a number of transformations.

```
module Tiger-ElimDead
imports Tiger Tiger-Rename
strategies
  eliminate-dead-functions =
    rec x(try(Let([VarDec(id,id,x)],id))
          ; {| IsDead :
                 try(DeclareDead)
               ; (Let([VarDec(id,id,id)] <+ x, x) <+ all(x))
               ; try(NotDead + Eliminate; try(RmEmptyLet))
              |})

  DeclareDead =
    Let([FunctionDec(map(DeclareDead))], id)

  DeclareDead =
    ?fdec@FunDec(f, _, _, _);
    rules( IsDead : FunDec(f,_,_,_) -> () )

  NotDead =
    ?Call(Var(f),_);
    override rules( IsDead : FunDec(f,_,_,_) -> Undefined )

  Eliminate =
    Let([FunctionDec(filter(not(IsDead)))],id)

  RmEmptyLet :
    Let([], e) -> e

  RmEmptyLet :
    Let([FunctionDec([])], e) -> e
```

Fig. 6. Strategy for eliminating dead functions

In an abstract interpretation style typechecker for Tiger, dynamic rules are used to generate typechecking rules for variables and functions. Thus, there is no need for threading type environments along traversals, and type rules can be expressed directly as rewrite rules.

In an interpreter for Tiger, dynamic rules are used to represent mappings from variables to values on the stack or heap. Variable bindings are dealt with using a scoped traversal similar to that of the renamer. Globally visible heap objects are represented by an unscoped dynamic rule that maps reference values (pointers) to values. Evaluation for individual constructs is expressed using constant folding rules.

Several optimizations for Tiger programs including constant propagation, copy propagation, and dead code elimination can be expressed elegantly using dynamic rules. In forward transformation problems, dynamic rules rewrite variables to constant or to copy expressions. In backward problems, dynamic rules keep track of use/def and neededness information. Instrumentations of Tiger programs for tracing and profiling use dynamic rules to selectively extend functions with extra functionality.

There are many other program transformations that can benefit from the use of dynamic rules. It seems that the data-flow transformations above can be easily extended to inter-procedural transformations by generating appropriate rules for function calls from their function declarations. The warm fusion algorithm for deforestation [11] uses dynamically generated rewrite rules for the derivation of catamorphisms from recursive function definitions. The implementation of warm fusion in Stratego [9] can be simplified using dynamic rules. Dynamic rules can also be used for memoization. The use/def analysis mentioned above uses a memoization scheme to incrementally recompute the analyses for an expression.

Another application area is the run-time configuration of transformation components. Options passed on the command-line can be used to generate rules used during a transformation. This can range from simple information such as optimization level to user-defined optimization rules and instantiation of an analysis with a set of initial variables to scrutinize.

## *6.2 Related Work*

### *6.2.1 Language Independent Traversals*

In [15] it is shown how generic traversal strategies can be used to define generic, language independent algorithms for language processing problems such as free variable extraction, bound variable renaming, substitution, and unification. These generic algorithms are parameterized with strategies for recognizing the various aspects of the object language such as representation of variables, variable binding constructs, and binding positions of binding constructs. Dynamic rules are orthogonal to generic traversals and can make their implementation easier since environment threading can be delegated to dynamic rules.

### *6.2.2 Assert in Prolog*

Dynamic rules are most closely related to the extra-logical operators assert and retract in Prolog. The goal `assert(G)` adds `G` to the rule database. All free logic variables in `G` are universally quantified. This is similar to the variables in dynamic rules that do not occur in the context. The goal `retract(G)` retracts from the rule database *all* rules that unify with `G`. The dynamic rule mechanism in this paper does not provide a retract. Instead, the live range of a rule can be controlled by means of rule scope that automatically retracts rules at the end of their scope. This provides a much cleaner way to retract rules, since only those rules generated before are retracted. Rules that were declared outside the scope become visible again. This cannot be modeled using retract.

A declarative formulation of assert in LProlog is described in [7].

### *6.2.3 Reflection*

Dynamic rules could be considered as a restricted form of computational reflection [13]. However, it is not nearly as general as general reflection in rewriting logic, as provided in Maude [6], which supports arbitrary manipulation of specifications at the meta-level at run-time. A reflective extension of ELAN is proposed in [10]. The more restricted form of reflection presented in this paper can be (and has been) effectively implemented by a compiler, i.e., does not require interpretation.

### *6.2.4 Dynamic Variables*

Dynamic rules are related to dynamically scoped variables in programming languages. In Lisp dynamic scope is considered a bug in the implementation. Domain-specific languages such as $\TeX$ makes use of dynamically scoped variables that allows for easy redefinition of behaviour; configuration of a document style can be influenced by redefining macros representing parameters of the style. More recently several papers [12,8] have reintroduced dynamic scope as a feature in general purpose languages.

Lewis et al. [12] introduce implicit parameters in functional languages such as Haskell. Implicit parameters can be used deeply embedded in a functional definition and can be bound at some outer level without having to pass the value explicitly through all the intermediate function calls. Hansson et al. [8] introduce dynamic variables in an imperative setting. A dynamic variable is created and initialized with an initial value. A use of a dynamic variable refers to the most recent setting of a dynamic variable with the same name. The main difference between the approaches is the fact that the value of an implicit parameter in [12] cannot be changed, while a dynamic variable is passed *by reference* to the use site and can thus be updated. Such updating corresponds to the notion of *overriding rules* generation in this paper.

In contrast to these approaches, dynamic rules define *mappings* from tuples of terms to tuples of terms (the bindings to context variables in the left-hand side and right-hand side, respectively). When introducing a new rule only the points in the mapping that overlap the left-hand side variables are shadowed, while other rule instances remain visible.

Furthermore, unlike the approaches of implicit parameters and dynamic variables, the dynamic rule *scope* construct in this paper is separated from both the generation and use of dynamic rules. This entails greater flexibility; strategies that introduce dynamic rules can be put to different use by manipulating the range of the scope. This is illustrated for example by the various strategies for variable renaming that reflect different (object language) scope rules, while (re-)using the same renaming rules.

### *6.3 Future Work*

Wadler's deforestation algorithm [18] can be expressed using rewrite rules and a simple strategy. Dynamic rules can be used to implement the folding of recursive occurrences of the function composition being deforested. However, this requires abstracting over object variables, which is not supported by the dynamic rules discussed in this papers. Currently, dynamic rules can only inherit ground terms from their definition context. Another application that would need abstraction over object variables are the rule pragmas of the Glasgow Haskell Compiler [14] that allow the user to state rewrite rules that should be applied during compilation in addition to normal optimizations.

In the scheme described in this paper each dynamic rule defines its own namespace. In order to achieve shadowing effects in the namespaces of other rules, these rules should be undefined. When mixing many rules this might become unattractive. Some means of declaring the namespace dependencies between rules will be useful.

Each of the Tiger transformations in this paper defines its own traversal over syntax trees and has to deal with the peculiarities of the scope rules for variable declarations. It would be better if the schema for the scope rules of a language could be captured in a generic strategy. However, this requires abstraction over dynamic rule names (rather than transformations) for limiting the scope of dynamic variables. This is possible in the underlying implementation, but it would be more attractive to express this at the level of the language extension.

It will also be interesting to investigate the interaction between various optimizations based on dynamic rules if they are combined in a single traversal.

Finally, dynamic scoping may give rise to unexpected behaviour when a dynamically generated rule `A` itself calls a dynamic rule `B`, which is intended to be the `A` *generation time* instance of rule `B`, instead of the `A` *call time* instance of `B`, i.e., `B` could change between the generation of `A` and a call to `A`. This can often be solved by invoking `B` in the generation context of `A`. However, in general this may require closures of dynamic rules.

## 7 Conclusion

This paper presented an extension of term rewriting with the run-time generation of context-dependent rewrite rules. Generated rules can be used as part of the global tree traversal, thus not increasing complexity by performing additional traversals. The extension is not limited to some specific form of program representation such as control flow graphs, but can be applied in the transformation of arbitrary abstract syntax trees.

Scoped dynamic rewrite rules solve (many of) the limitations caused by the context-free nature of rewrite rules, strengthening the separation of rules and strategies, and supporting concise and elegant specification of program transformations. This has been illustrated in this paper by the specification of three transformations, i.e., bound variable renaming, function inlining, and dead function elimination.

**Acknowledgments**

I would like to thank Patricia Johann for comments on a previous version of this paper.

## References

[1] A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[2] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, September 1997.

[3] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier.

[4] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transforming system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 353–372. Birkhäuser, 1997.

[5] M. G. J. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, Centrum voor Wiskunde en Informatica, 2001.

[6] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First International Workshop on Rewriting Logic and its Applications.

[7] S. Dietzen and F. Pfenning. A declarative alternative to “assert” in logic programming. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 372–386, San Diego, USA, 1991. The MIT Press.

[8] D. R. Hanson and T. A. Proebsting. Dynamic variables. In *Programming Language Design and Implementation (PLDI'01)*, Snowbird, UT, USA, June 2001. ACM.

[9] P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):1–34, 2000. .

[10] H. Kirchner and P.-E. Moreau. A reflective extension of ELAN. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First International Workshop on Rewriting Logic and its Applications.

[11] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In S. L. P. Jones, editor, *Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 314–323. ACM Press, June 1995.

[12] J. R. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 108–118. ACM, January 2000.

[13] P. Maes. Concepts and experiments in computational reflection. *SIGPLAN Notices*, 22(12):147–155, 1987. Proceedings of OOPSLA'87.

[14] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In R. Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, September 2001. ACM SIGPLAN.

[15] E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht. .

[16] E. Visser. A survey of strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57/2 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers. .

[17] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98) .

[18] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.